www.ijecs.in
International Journal Of Engineering And Computer Science ISSN:2319-7242
Volume 4 Issue 3 March 2015, Page No. 10791-10798

Analysis of Use Cases and Use Case Estimation

[1.] Abhishek Chaudhary [2.] Nalin Chaudhary [3.] Aasiya Khatoon
Assistant Professor (C.S.E)
M.Tech Scholar (C.S.E)
M.Tech Scholar (C.S.E)
Bhagwant University, Ajmer
abhishek02mar@rediffmail.com nalin23jan1990@gmail.com ashi.shiekh@gmail.com

Abstract- Use case analysis is a major technique used to find out the functional requirements of a software system. Use case, an important concept in use case analysis, represents an objective user wants to achieve with a system. It can be in text form, or be visualized in a use case diagram. There are different approaches and methods to successfully estimate effort using use cases. This Paper describes use cases and how to write them, and presents the Use Case Points method. It also describes related work on estimating with use cases.
Keywords: Use case, Use case point methods, software project estimation.

I. INTRODUCTION

The term 'use case' implies 'the ways in which a user uses a system'. It is a collection of possible sequences of interactions between the system under construction and its external actors, related to a particular goal. Actors are people or computer systems, and the system is a single entity, which interacts with the actors [1].

The purpose of a use case is to meet the immediate goal of an actor, such as placing an order. To reach a goal, some action must be performed [2]. All actors have a set of responsibilities. An action connects one actor's goal with another's responsibility [3].

A primary actor is an actor that needs the assistance of the system to achieve a goal. A secondary actor supplies the system with assistance to achieve that goal. When the primary actor triggers an action, calling up the responsibilities of the other actor, the goal is reached if the secondary actor delivers [3].

1. The Graphical Use Case Model

The use case model is a set of use cases representing the total functionality of the system. A complete model also specifies the external entities such as human users and other systems that use those functions. UML provides two graphical notations for defining a system functional model:

- The use case diagram depicts a static view of the system functions and their static relationships with external entities and with each other. Stick figures represent the actors, and ellipses represent the use cases. See figure 1.
- The activity diagram imparts a dynamic view of those functions.

The use case model depicted in Figure 1 is the model of an hour registration system. The user enters user name and password, is presented with a calendar and selects time periods, and then selects the projects on which to register hours worked.

2. Scenarios and Relationships

A scenario is a use case instance, a specific sequence of actions that illustrates behaviors. A main success scenario describes what happens in the most common case when nothing goes wrong. It is broken into use case steps, and these are written in natural language or depicted in a state or an activity diagram [4].

Different scenarios may occur, and the use case collects together those different scenarios [1].

Use cases can include relationships between themselves. Since use cases represent system functions, these relationships indicate corresponding relationships between those system functions. A use case may either always or sometimes include the behaviour of another use case; it may use either an 'include' or an 'extend' relationship. Common behaviour is factored out in included use cases. Optional sequences of events are separated out in extending use cases.

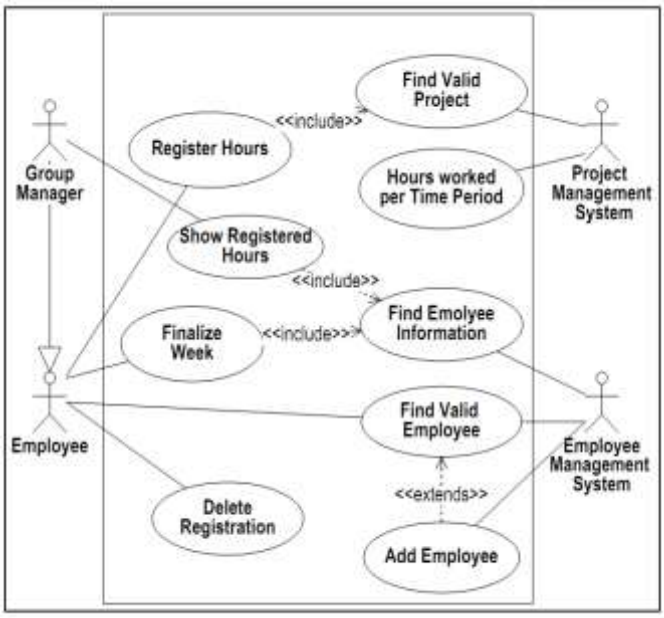

Figure 1: A graphical use case model

3. Generalization between Actors

A clerk may be a specialization of an employee, and an employee may be a generalization of a clerk and a group manager, see Figure 2 on the next page. Generalizations are used to collect together common behaviour of actors.

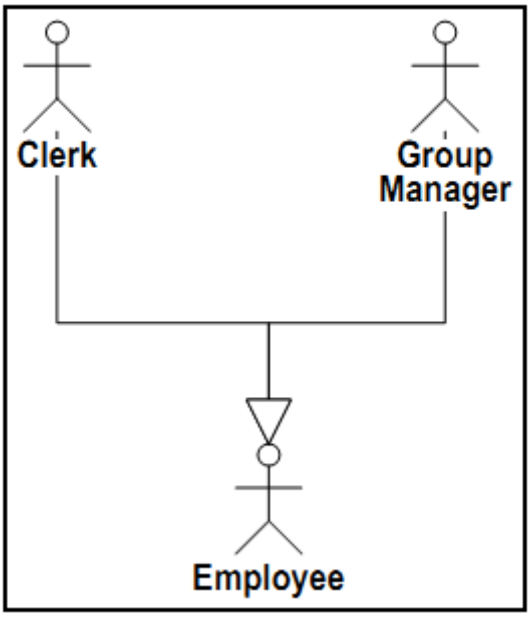

Figure 2: Generalization between actors

II. The Use Case Points Method

An early estimate of effort based on use cases can be made when there is some understanding of the problem domain, system size and architecture at the stage at which the estimate is made [5]. The use case points method is a software sizing and estimation method based on use case counts called use case points.

1. Classifying Actors and Use Cases

Use case points can be counted from the use case analysis of the system. The first step is to classify the actors as simple, average or complex. A simple actor represents another system with a defined Application Programming Interface, API, an average actor is another system interacting through a protocol such as TCP/IP, and a complex actor may be a person interacting through a GUI or a Web page. A weighting factor is assigned to each actor type.

- Actor type: Simple, weighting factor 1
- Actor type: Average, weighting factor 2
- Actor type: Complex, weighting factor 3

The total unadjusted actor weights (UAW) is calculated by counting how many actors there are of each kind (by degree of complexity), multiplying each total by its weighting factor, and adding up the products. Each use case is then defined as simple, average or complex, depending on number of transactions in the use case description, including secondary scenarios. A transaction is a set of activities, which is either performed entirely, or not at all. Counting number of transactions can be done by counting the use case steps. Use case complexity is then defined and weighted in the following manner:

- Simple: 3 or fewer transactions, weighting factor 5
- Average: 4 to 7 transactions, weighting factor 10
- Complex: More than 7 transactions, weighting factor 15

Another mechanism for measuring use case complexity is counting analysis classes, which can be used in place of transactions once it has been determined which classes implement a specific use case [6]. A simple use case is implemented by 5 or fewer classes, an average use case by 5 to10 classes, and a complex use case by more than ten classes. The weights are as before. Each type of use case is then multiplied by the weighting factor, and the products are added up to get the unadjusted use case weights (UUCW).

The UAW is added to the UUCW to get the **unadjusted use case points UUPC):**

UAW+UUCW=UUCP

2. Technical and Environmental Factors

The method also employs a technical factors multiplier corresponding to the Technical Complexity Adjustment factor of the FPA method, and an environmental factors multiplier in order to quantify non-functional requirements such as ease of use and programmer motivation.

Various factors influencing productivity are associated with weights, and values are assigned to each factor, depending on the degree of influence. 0 means no influence, 3 is average, and 5 means strong influence throughout. See Table 1 and Table 2.

The adjustment factors are multiplied by the unadjusted use case points to produce the adjusted use case points, yielding an estimate of the size of the software. The Technical Complexity Factor (TCF) is calculated by multiplying the value of each factor (T1- T13) by its weight and then adding all these numbers to get the sum called the T-Factor. The following formula is applied:

TCF = 0.6 + (0.01* T-Factor)

The Environmental Factor (EF) is calculated by multiplying the value of each factor (F1-F8) by its weight and adding the products to get the sum called the E-Factor. The following formula is applied:

EF= 1.4 + (-0.03 * E-Factor)

The adjusted use case points (UPC) are calculated as follows:

UPC= UUCP*TCF*EF

3. Problems with Use Case Counts

There is no published theory for how to write or structure use cases. Many variations of use case style can make it difficult to measure the complexity of a use case [7]. Free textual descriptions may lead to ambiguous specifications [8]. Since there is a large number of interpretations of the use case concept, Symons concluded that one way to solve this problem was to view the MkII logical transaction as a specific case of a use case, and that using this approach leads to requirements which are measurable and have a higher chance of unique interpretation.

Factor	Description	Weight
T1	Distributed System	2
T2	Response adjectives	2
T3	End-user efficiency	1
T4	Complex processing	1
T5	Reusable code	1
T6	Easy to install	0.5
T7	Easy to use	0.5
T8	Portable	2
T9	Easy to change	1
T10	Concurrent	1
T11	Security features	1
T12	Access for third parties	1
T13	Special training required	1

Table 1: Technical Complexity Factors

Factor	Description	Weight
F1	Familiar with RUP	1.5
F2	Application experience	0.5
F3	Object-oriented experience	1
F4	Lead analyst capability	0.5
F5	Motivation	1
F6	Stable requirements	2
F7	Part-time workers	-1
F8	Difficult programming language	2

Table 2: Environmental Factors

III. WRITING USE CASES

The use cases of the system under construction must be written at a suitable level of detail. It must be possible to count the transactions in the use case descriptions in order to define use case complexity. The level of detail in the use case descriptions and the structure of the use case have an impact on the precision of estimates based on use cases. The use case model may also contain a varying number of actors and use cases, and these numbers will again affect the estimates [9].

- **The Textual Use Case Description**

The details of the use case must be captured in textual use case descriptions written in natural language, or in state or activity diagrams. A use case description should at least contain an identifying name and/or number, the name of the initiating actor, a short description of the goal of the use case, and a single numbered sequence of steps that describe the main success scenario [4].The main success scenario describes what happens in the most common case when nothing goes wrong. The steps are performed strictly sequentially in the given order. Each step is an extension point from where alternative behaviour may start if it is described in an extension. The use case model in Figure 1 is written out as follows:

Use Case Descriptions for Hour Registration System

Use case No. 1
Name: Register Hours
Initiating Actor: Employee
Secondary Actors: Project Management System
Employee Management System
Goal: Register hours worked for each employee on all projects the employee participates on
Pre-condition: None

MAIN SUCCESS SCENARIO

1. The System displays calendar (Default: Current Week)
2. The Employee chooses time period
3. Include Use Case 'Find Valid Projects'
4. Employee selects project
5. Employee registers hours spent on project
   Repeat from 4 until done
6. The System updates time account

EXTENSIONS

2a. Invalid time period
The System sends an error message and prompts
user to try again

This use case consists of 6 use case steps, and one extension step, 2a.Step 2 acts as an extension point. If the selected time period is invalid, for instance if the beginning of the period is after the end of the period, the system sends an error message, and the user is prompted to enter a different period. If the correct time period is entered, the use case proceeds. The use case also includes another use case, 'Find Valid Projects'. This use case is invoked in step 3. When a valid project is found by the Project Management System, it is returned and the use case proceeds. The use case goes into a loop in step 5, and the employee may register hours worked for all projects he/she has worked on during the time period. The use case 'Find Valid Employee' is extended by the use case 'Add Employee'.

Use case No. 2
Name: Find Valid Employee
Initiating Actor: Employee
Secondary Actor: Employee Management System
Goal: Check if Employee ID exists
Pre-condition: None

MAIN SUCCESS SCENARIO

1. Employee enters user name and password
2. Employee Management System verifies user name and password
3. Employee Management System returns Employee ID

EXTENSIONS

2a. Error message is returned
2b. Use Case 'Add Employee'

IV. RELATED WORK

Different methods for sizing object-oriented software projects and computing estimates of effort have been proposed over the last years. Some of these methods are presented in the following.

1. Mapping Use Cases into Function Point Analysis

A method for mapping the object-oriented approach into Function point analysis is described by Thomas Fetke et al., [10]. The authors propose mapping the use cases directly into the Function point model using a set of concise rules that support the measurement process. These mapping rules are based on the standard FPA defined in the IFPUG Counting Practices manual. Since the concept of actors in the use case model is broader than the concept of users and external applications in FPA, there cannot be a one-to-one mapping of actors and users to external applications. But each user of the system is defined as an actor. In the same manner, all applications which communicate with the system under consideration must also appear as actors. This corresponds to Karner's use case point method.

The level of detail in the use case model may vary, and the use case model does not provide enough information to how to count a specific use case according to function point rules. Therefore, as in Karner's method, the use cases must be described in further detail in order to be able to count transactions.

2. Use Case Estimation and Lines of Code

John Smith of Rational Software describes a method presenting a framework for estimation based on use cases translated into lines of code [7]. There does not seem to be any more research done on this method, although the tool 'Estimate Professional', which is supplied by the Software Productivity Center Inc, and the tool 'CostXpert' from Marotz Inc. produce estimates of effort per use case calculated from the number of lines of code.

3. Use Cases and Function Points

David Longstreet of Software Metrics observed that applying function points helps to determine if the use case is written at a suitable level of detail [11]. If it is possible to describe how data passes from the actor to inside the boundary or how data flows from inside the application boundary to the actor, then that is the right level of detail,

otherwise the use case needs more detail. By adopting both the use case method and the function point's method, the quality of the requirement documents can be improved. Thus, sizing and estimating is improved.

4. The COSMIC-FFP Approach

Over the last 15 years or so, advances have been made towards a general Functional Size Measurement (FSM) method for measuring real-time software. Recently, the COSMIC FFP (Full Function Points) method has been developed as an improvement of the earlier function point methods. It is designed to work for both business applications and real-time software [12].

When sizing software using the traditional function point methods, it is possible to measure only the functionality as seen by the human end-user.

The large amounts of functionality that must be developed in today's advanced software systems are invisible to the users and cannot be measured by these methods. Using the traditional methods may correctly size the functionality seen by the user, but grossly undersize the total functionality that actually has to be developed.

The Full Function Points (FFP) methodology is a functional size measurement technique specifically designed to address the requirements of embedded and real-time software. The FFP methodology is based on a 'unit of software delivered' metric called the FFP point, which is a measure of the functional size of the software. The total FFP point of an application being measured is called an FFP count.

Functional user requirements are decomposed into `functional processes' which in turn can be decomposed into `functional sub-processes'. The functional processes are equivalent to the MKII logical functions and also to use cases. The method can therefore be used to size object-oriented software.

V. CONCLUSION

This paper looks at the potential of successful application of the use case point method for estimating the size of software development project. A use case point is a new method for estimating software development. Advantage of the use case based estimation is that use cases are maintained with two-way traceable capability using modern requirements management tools. In conclusion, use case points method of effort estimation is a very valuable addition to the tools available for the project manager. The method can be very reliable or just as reliable as other effort estimation tools such as COCOMO, function point and lines of code.

REFRENCES

[1] Alistair Cockburn. Writing Effective Use Cases. Addison-Wesley, 2000.

[2] Charles Richter. Designing Flexible Object-Oriented Systems with UML. Macmillan Technical Publishing, 2001.

[3] Alistair Cockburn. Structuring use cases with goals. Humans and Technology, 1997.

[4] John Cheesman and John Daniels. UML Components, A simple Process for Specifying Component-based Software. Addison-Wesley, 2000.

[5] Steve Sparks and Kara Kaspczynski. The art of sizing projects. Sun World.

[6] Schneider and Winters. Applying use Cases. Addison-Wesley,1998.

[7] John Smith. The estimation of effort based on use cases. Rational Software White Paper, 1999.

[8] Martin Arnold and Peter Pedross. Software size measurement and productivity rating in a large-scale software development department. Forging New Links. IEEE Comput. Soc, Los Alamitos, CA,USA, 1998.

[9] Bente Anda, Hege Dreiem, Magne Jørgensen, and Dag Sjøberg. Estimating Software Development Effort based on Use Cases - Experience from Industry. In M. Gogolla, C. Kobryn (Eds.): UML 2001- The Unified Modeling Language. Springer-Verlag. 4th International Conference, Toronto, Canada, October 1-5, 2001, LNCS 218, 2001.

[10] Thomas Fetke, Alan Abran, and Tho-Hau Ngyen. Mapping the oo-jacobsen approach into function point analysis. The Proceedings of TOOLS, 23, 1997.

[11] David Longstreet. Use cases and function points. Copyright Long street Consulting Inc. www.softwaremetrics.com, 2001.

[12] P.Grant Rule. Using measures to understand requirements. Software Measurement Services Ltd, 2001.